AD/A-002 279

# AN INTERACTIVE PROGRAM VERIFICATION SYSTEM

Donald I. Good, et al

University of Southern California

Prepared for:

Advanced Research Projects Agency
National Science Foundation
Texas University

October 1974

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER ISI/RR-74-22 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER AD/A-002279 |
| 4. TITLE (and Subtitle) AN INTERACTIVE PROGRAM VERIFICATION SYSTEM | | 5. TYPE OF REPORT & PERIOD COVERED Research Report |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s) Donald I. Good, U of Texas at Austin & ISI; Ralph L. London, ISI; W. W. Bledsoe, U of Texas at Austin | | 8. CONTRACT OR GRANT NUMBER(s) DAHC 15 72 C 0308 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, California 90291 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order No. 2223 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, Virginia 22209 | | 12. REPORT DATE October 1974 |
| | | 13. NUMBER OF PAGES 31 |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | | 15. SECURITY CLASS. (of this report) None |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

Distribution unlimited. Available from National Technical Information Service, Springfield, Virginia 22151.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Program verification, theorem proving, program proving, correctness, simplification, reliable software, interactive system.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## ABSTRACT

This paper is an initial progress report on the development of an interactive system for verifying that computer programs meet given formal specifications. The system is based on the conventional inductive assertion method: given a program and its specifications, the object is to generate the verification conditions, simplify them, and prove what remains. This system addresses two aspects of software improvement of extreme importance to the military: increase in the quality of software and decrease in the cost of producing high-quality software. The important feature of the system is that the human user has the opportunity and obligation to help actively in the simplifying and proving. The user, for example, is the primary source of problem domain facts and properties needed in the proofs. A general description is given of the overall design philosophy, structure, and functional components of the system, and a simple sorting program is used to illustrate both the behavior of major system components and the type of user interaction the system provides.

ARPA ORDER NO. 2223

ISI/RR-74-22
Oct[illegible]r 1974

**Donald I. Good**
University of Texas at Austin
and USC/Information Sciences Institute

**Ralph L. London**
USC/Information Sciences Institute

**W. W. Bledsoe**
University of Texas at Austin

# An Interactive Program Verification System

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

This research is wholly supported at USC Information Sciences Institute by the Advanced Research Projects Agency under Contract DAHC15 72 C 0308 and is supported at the University of Texas at Austin by the National Science Foundation under Grants GJ-36424 and GJ-32269. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the sponsors or the United States Government.

## *ABSTRACT*

This paper is an initial progress report on the development of an interactive system for verifying that computer programs meet given formal specifications. The system is based on the conventional inductive assertion method: given a program and its specifications, the object is to generate the verification conditions, simplify them, and prove what remains. This system addresses two aspects of software improvement of extreme importance to the military: increase in the quality of software and decrease in the cost of producing high-quality software. The important feature of the system is that the human user has the opportunity and obligation to help actively in the simplifying and proving. The user, for example, is the primary source of problem domain facts and properties needed in the proofs. A general description is given of the overall design philosophy, structure, and functional components of the system, and a simple sorting program is used to illustrate both the behavior of major system components and the type of user interaction the system provides.

## *ACKNOWLEDGMENTS*

Many people have contributed in various ways to the development of this system. We are especially indebted to Raymond Bates, Peter Bruell, Lawrence Fagan, Anthony Hearn, Shigeru Igarashi, David Luckham, Donald Lynn, David Musser, and Mabry Tyson.

## *INTRODUCTION*

In many computer application areas, the consequences of a program not performing as intended can be quite costly or damaging. The goal of this research is to develop an interactive system for proving that programs are consistent with precisely stated specifications. A proved program, therefore, will always perform correctly if the specifications are an adequate reflection of the actual intent of the program. The purpose of the verification system described here is to provide an integrated set of automatic and interactive facilities that make it possible to construct a proof that a program is consistent with given specifications. The human user of the system, rather than the system itself, retains primary responsibility for the proof of his program. It is assumed that the user is highly knowledgeable in the domain of the problem that his program is to solve, in programming, and in the methods of proof supported by the system.

Previous experience with actual proofs of a variety of programs and with various experimental program verification systems seems to support three preliminary conclusions. First, proofs of significant programs are possible. Second, automatic program verifiers can be of considerable help in proving real programs. Third, program proof methods can best be exploited if the proof can be approached on a segment-by-segment basis that reflects the structure of the program. Probably the single example that most strongly supports these conclusions is the proved verification condition generation program of Ragland [1973]. The entire program consists of 203 procedures, most of these occupying less than one page including assertions. Two factors contributed significantly to the completion of this proof. First, the program was written as a large collection of small procedures so that the proof could be done one procedure at a time. Second, an automatic verification condition generator [Wang 1973] was used to construct the lemmas that were sufficient for a proof. Although all of these lemmas were proved manually, a subjective evaluation indicated that about 70% of them were sufficiently simple to have been proved automatically.

Further support for the three preliminary conclusions is provided by the achievements of several existing program verification systems (see Waldinger and Levitt [1973] and references therein). Programs such as sorting, pattern matching, unification, array rearrangement, and finding primes via a sieve have been verified with various amounts of effort, either with no human assistance or with very little assistance. The statement by Waldinger and Levitt [1973, p. 169] that "there is no shortage of interesting work related to our own" also applies to us.

This paper is an initial progress report on the development of a verification system whose ultimate goal is to be an effective tool in constructing rigorous proofs of significant programs. The design philosophy of the system is based on the belief that large parts of the total proof of actual programs can, and should, be done automatically, but also that, in the foreseeable future, some parts will have to be done by humans. This seems a proper response to the genuinely open-ended nature of facts, theorems, and deductions needed to verify realistic programs. Thus our design strategy has been to provide automatic capability for the proof process where practical, and to rely on interaction for manual intervention otherwise. If a program can be verified with no human assistance, then we shall applaud the system's achievement. We will be quite pleased, however, if the system provides sufficient assistance so that the verification can be completed with minimal human hints or proof steps.

## *SYSTEM STRUCTURE*

The system is based on the conventional inductive assertion method of proving properties of programs, and consists of five major components: a standard text editor, a program and assertion parser for Pascal programs, a verification condition generator, a simplification and substitution package, and an interactive theorem prover. Some preliminary work has also been done on an interpreter that is capable of running programs on both actual and symbolic data, but the interpreter has not yet been exploited in any significant way. The entire system is Lisp-based and runs as a large program on a PDP-10 computer. The system is implemented primarily in Reduce [Hearn 1971], a Lisp-based symbolic mathematical system, but certain components are written directly in Stanford Lisp 1.6.

One of the unique features of this system is the extent to which we have been able to use previously written and highly developed programs as major system components. First of all, Reduce, in addition to its powerful, well-developed algebraic manipulation capability, has served as an effective language for system implementation, and will permit the system to be as portable as Reduce itself. Our PDP-10 implementation of Reduce also has a built-in link to a text editor, and this provides the editor for the verification system. The Pascal parser, developed at USC Information Sciences Institute, was written in Reduce. The verification condition generator is essentially the Pascal generator of Igarashi, London, and Luckham [1973], originally developed at Stanford but now existing in Lisp 1.6. The simplification and substitution package was developed at ISI, drawing, in part, on the algebraic manipulation capability of Reduce. The theorem prover is a variation of the prover described by Bledsoe and Bruell [1973]. The prover originally was developed at the University of Texas at Austin in UT-Lisp, and was translated into Stanford Lisp 1.6 for incorporation into the system. Although the prover has been modified to make it more effective on the types of theorems encountered in proving programs, its basic structure and interactive philosophy remain valid and unchallenged.

Currently, the normal mode of using the system is to invoke interactively the system components in the following sequence: create the program and specifications, parse them checking for syntax errors, generate verification conditions, simplify them, and prove those that do not simplify to TRUE. The user also can descend directly into Reduce or Lisp. The following sections describe the main ideas of each of the major system components and illustrate their use in proving the sorting program shown in Figure 1.

```
FUNCTION LOCMAX(A:INTARRAY;I,J:INTEGER):INTEGER;
ENTRY I LE J;
EXIT  (I LE LOCMAX(A,I,J))
  AND (LOCMAX(A,I,J) LE J)
  AND (A[LOCMAX(A,I,J)] = AMAX(A,I,J));;


FUNCTION SORT(A:INTARRAY;N:INTEGER):INTARRAY;
ENTRY N GE 1;
EXIT  ALL I ( (1 LE I) AND (I LE N) IMP
        SORT(A,N)[I] = AMAX(SORT(A,N),1,I))
  AND APERM(SORT(A,N),A,1,N);
VAR B:INTARRAY;
BEGIN
  B := A;
  K := N;
  ASSERT ALL I ((K+1 LE I) AND (I LE N) IMP B[I] = AMAX(B,1,I))
    AND APERM(B,A,1,N)
    AND (K GE 1) AND (K LE N);
  WHILE K > 1 DO
    BEGIN
      B := ASWAP(B, LOCMAX(B,1,K), K);
      K := K - 1;
    END;
  SORT := B;
END; .
```

Figure 1. Sample sorting program with specifications.

## *PROGRAMS TO BE VERIFIED*

The system proves assertions that have been inserted into Pascal programs. The specific choice of Pascal is not an essential point; what matters is that the language features adopted in Pascal seem important and (with some exceptions, of course) representative of those in commonly-used programming languages. We believe that extensive subsets of other languages could be made a part of the verification system by using an appropriate parser, and either writing an additional verification condition generator or modifying the current one.

The language subset of Pascal considered so far allows programs to be constructed from the following syntactic units: assignment, conditional, while, repeat, for, compound, go-to, and null statements; recursive procedure and function definitions and calls; one-dimensional arrays; arithmetic, relational, and Boolean expressions; and labels. Pascal has been extended to allow certain types of operations that are not ordinarily possible. For example, in the program in Figure 1 the functions SORT and ASWAP each return complete integer arrays as values; furthermore, array-valued assignments such as B := ASWAP(B,LOCMAX(B,1,K),K) are permitted. These operations are a simple means of introducing modularity and abstraction into a program. This type of programming is discussed in more detail by Good [1974].

A second extension to Pascal allows ENTRY, EXIT, and ASSERT statements, which are the means of stating the specifications to be proved about the program. A proof shows that the ENTRY assertion always implies the EXIT assertion if the program terminates. The ASSERT statements supply the intermediate assertions for a proof by inductive assertions, and each loop in a program must contain at least one ASSERT statement. To ensure this the system requires an ASSERT statement to appear (a) before every WHILE and before every FOR statement, (b) as either the first or last statement of the repeated statements in a REPEAT statement, and (c) after every label. Otherwise, the placement of assertions is optional. (The assertion before a WHILE or FOR statement is considered to be just before the "test" in each case.) The expressions following the ENTRY, EXIT, and ASSERT statements are Boolean-valued expressions of Pascal augmented by implication, logical equivalence, and quantification. Because function calls are permitted in expressions (which thereby also permits arbitrary predicates with appropriate parameters), this assertion language allows us to express, in principle, all that is needed. In practice it is somewhat limited, nontransparent, and inelegant, but not overly burdensome. Extensions and additional notations are planned.

In the SORT function example the EXIT assertion specifies a sort into ascending order. The function AMAX(A,I,J) denotes the value of a maximum element in the array segment A[I . . . J], and APERM(A,B,I,J) states that array segment A[I . . . J] is a permutation of B[I . . . J]. The system has no built-in knowledge about these functions, and appropriate facts about them will be supplied interactively during the proof. The system does have some built-in simplification rules for ASWAP(A,I,J) which swaps elements I and J in array A. Notice also, in this example, that ENTRY and EXIT assertions have been stated for the user-supplied function LOCMAX without giving its actual code. In proving SORT, all we need to know about LOCMAX(A,I,J) is the specification that it returns the *location* of a maximum element in the array segment A[I . . . J] (which specification we have separately proved for several versions of code for LOCMAX). This facility permits the top-down design and top-down proof of programs.

## *VERIFICATION CONDITION GENERATOR*

The verification condition generator is an implementation of the axioms and rules of inference which constitute the axiomatic definition of Pascal (Hoare and Wirth [1973], and Hoare [1971]). By invoking these semantic rules, the consistency question between program and specifications is reduced to proving a set of mathematical lemmas sufficient to show that the ENTRY assertion of the program always implies the EXIT assertion.

The use of an axiomatic definition of a programming language as the basis of a verification condition generator has been described with numerous examples by Igarashi, London, and Luckham [1973]. Essentially, the idea is to implement the axioms and rules of inference in such a way that for each type of program statement, exactly one axiom or rule of inference is applicable to the type. It is then possible to generate recursive subgoals deterministically, i.e., to compute without search, sufficient lemmas to imply the desired properties about the program. For example, the ENTRY assertion of SORT always implies the EXIT assertion if the three lemmas in Figure 2 are satisfied. These verification conditions are the actual output of the verification condition generator.

Verification condition generators have, of course, been constructed in other ways. Both the present subgoaling and the backward substitution can be replaced by various methods. Furthermore, other formalisms besides axiomatic definitions, such as state vector approaches, can be the basis for generating the lemmas. Although the results of the various methods may be superficially different in appearance, the results are logically equivalent, as, of course, they must be.

There is also an alternative verification condition generator in the system. Written by Musser and based on forward symbolic evaluation, it processes certain Reduce programs. The parsing of these Reduce programs is done by the regular Reduce translator. Further details and examples are in London and Musser [1974].

```
VC1:

    N GE 1
 IMP      ALL I ((N+1 LE I) AND (I LE N) IMP A[I] = AMAX(A, 1, I))
      AND APERM(A, A, 1, N)
      AND N GE 1
      AND N LE N
```

```
VC2:
          ALL I ((K+1 LE I) AND (I LE N) IMP B[I] = AMAX(B, 1, I))
      AND APERM(B, A, 1, N)
      AND K GE 1
      AND K LE N
      AND K>1
 IMP      1 LE K
     AND            (1 LE LOCMAX(B, 1, K)) AND (LOCMAX(B, 1, K) LE K)
               AND B[LOCMAX(B, 1, K)] = AMAX(B, 1, K)
          IMP      ALL I (     ((K-1) + 1 LE I) AND (I LE N)
                          IMP   ASWAP(B, LOCMAX(B, 1, K), K)[I]
                                 = AMAX(ASWAP(B, LOCMAX(B, 1, K), K), 1, I))
               AND APERM(ASWAP(B, LOCMAX(B, 1, K), K), A, 1, N)
               AND K-1 GE 1
               AND K-1 LE N
```

```
VC3:
          ALL I ((K+1 LE I) AND (I LE N) IMP B[I] = AMAX(B, 1, I))
      AND APERM(B, A, 1, N)
      AND K GE 1
      AND K LE N
      AND NOT (K>1)
 IMP      ALL I ((1 LE I) AND (I LE N) IMP B[I] = AMAX(B, 1, I))
      AND APERM(B, A, 1, N)
```

Figure 2. Verification conditions (lemmas) for SORT in Figure 1.

## *SIMPLIFICAT'ᴺN AND SUBSTITUTION*

The first step in proving the verification conditions is the application of a simplification and substitution package. Each verification condition is assigned a unique name and is processed individually under interactive control. The package consists of (a) a symbolic evaluator that applies reduction rules to expressions within the verification condition and (b) an interactively controlled substitution routine that is triggered by equalities in the main hypotheses of the verification condition. Working together, the evaluator and substitution routine achieve many of the simple proofs and reductions that typically arise in proving programs.

Symbolic evaluation is performed on the integer-valued operators + (n-ary), - (unary and binary), * (n-ary), DIV, MOD, EXPT (power), MAX (n-ary), and MIN (n-ary). Expressions are represented in prefix form throughout the evaluation, and each is evaluated to a standard form. Associative operators are represented in n-ary form, and the arguments of commutative operators are placed in a standard order. For example, both MAX(B,MAX(A,C)) and MAX(C,A,B) become the prefix expression (MAX A B C). The algebraic manipulation also collects constant terms insofar as possible, so that, for example, (K+1) -1 reduces to K. This is the kind of simple reduction that can remove a great deal of the clutter often found in verification conditions.

The evaluator also has a limited capability for symbolic manipulation of arrays. Currently, there are three operations involving arrays, all written as functions: referencing an array element (ASUB), changing the value of a single element (ASET), and swapping two elements (ASWAP). Examples of array reductions are

ASUB(ASET(A,I,X),I) reduces to X

and

ASUB(ASWAP(A,I,J),I) reduces to ASUB(A,J)

where ASUB(A,I) denotes A[I] and ASET(A,I,X) denotes A after doing A[I] := X.

Normally, most of the symbolic evaluation of a verification condition is involved with Boolean-valued operators. These include AND (n-ary), OR (n-ary), NOT, IMP (implies), EQV (logical equivalence), SOME (there exists), and ALL (for all), as well as the relational operators < (less than), LE (less than or equal), > (greater than), GE (greater than or equal), =, and NE (not equal). Any expression of the form "x operator y" whose operator is <, LE, >, or GE is reduced to an equivalent prefix form (LE 0 z). Equalities x = y are reduced to (= 0 x-y) and similarly for NE. The Boolean AND operator checks for inconsistencies among conjunctions of relational expressions, eliminates redundancies, and converts certain inequalities to equalities. For example, (N < I) AND (I LE N+1) is converted to I = N+1. The evaluator currently does not perform any kind of transitivity analysis on the relational expressions.

The evaluator reduces verification conditions (and implications in general) to the form (H1 AND . . . AND Hn) IMP (C1 AND . . . AND Cm). Once in this form, conclusions that match hypotheses are eliminated automatically. Conclusions that are relational expressions are negated and ANDed into the hypotheses. If a contradiction is attained, the relational conclusion is proved and consequently eliminated. Equalities that are hypotheses in the top-level implication of the verification condition invoke possible substitutions. Certain types of equalities cause automatic substitution throughout the verification condition, while others allow for interactive control of the substitution. An example is in VC2 of SORT where the substitution

SUB: B[LOCMAX(B,1,K)] := AMAX(B,1,K) ?

is proposed. The user may respond to go ahead with the substitution as proposed, reverse its direction, not do it, or examine the equality for other possible substitutions. The user may also specify that the substitution be made in hypotheses only or conclusions only. Substitutions in the hypotheses cause the equality that triggered the substitution to be eliminated.

Because substitutions may require interaction, it is important that a smooth user-system interface be provided. Toward this end, the evaluator and the substitution routine have been designed to give an optional running commentary of their operation so that the user can be kept in proper context. Considerable effort has been made to print expressions in a readable, pleasing form. Also, a special optional CRT display package and interaction capability has been developed for observation and interaction. The CRT screen is split into two parts, one containing the expression being processed and the other a scrolling workspace for the user-system dialogue. The evaluator and the substitution routine are also designed so that they can be manually interrupted and restarted or redirected.

The simplified verification conditions SVC1, SVC2, and SVC3 for SORT are shown in Figure 3. These represent a useful simplification over the originals. (In SVC2 the interactive response to the proposed substitution was NO.) It is these simplified verification conditions that are passed on to the theorem prover.

```
SVC1:
    1 LE N
 IMP APERM(A, A, 1, N)



SVC2:
(H1)          1 LE LOCMAX(B, 1, K)
(H2)       AND LOCMAX(B, 1, K) LE K
(H3)       AND K LE N
(H4)       AND 2 LE K
(H5)       AND APERM(B, A, 1, N)
(H6)       AND ALL I ((I LE N) AND (K<I) IMP B[I] = AMAX(B, 1, I))
(H7)       AND B[LOCMAX(B, 1, K)] = AMAX(B, 1, K)
(C1) IMP       APERM(ASWAP(B, LOCMAX(B, 1, K), K), A, 1, N)
(C2)       AND ALL I (     (I LE N) AND (K LE I)
                IMP   ASWAP(B, LOCMAX(B, 1, K), K)[I]
                   = AMAX(ASWAP(B, LOCMAX(B, 1, K), K), 1, I))



SVC3:
        APERM(B, A, 1, N)
     AND ALL I ((I LE N) AND (K<I) IMP B[I] = AMAX(B, 1, I))
     AND K LE N
     AND 1=K
 IMP ALL I ((I LE N) AND (1 LE I) IMP B[I] = AMAX(B, 1, I))
```

Figure 3. Simplified verification conditions for SORT.

It should be pointed out that there is currently some significant overlap between the tasks performed by the simplification package and those done by the theorem prover. An appropriate division between these two components remains a subject for further study. Also, it is worth noting that the Pascal verification condition generator does not use the simplification package during the generation process, as is done by other verifiers and also by Musser's verification condition generator. Interestingly enough, the separation of these two components was originally expected to cause problems, but so far it has not.

## *THEOREM PROVER*

Those verification conditions that do not simplify to TRUE are passed on to the interactive theorem prover. Briefly speaking, the prover is a natural deduction system that proves theorems by subgoaling (splitting), matching, and rewriting. It also utilizes semantic tables to help direct its search. The theorems (and subsequent subgoals) are shown on the user terminal in a natural, easy to read form, and the user is provided with several interactive commands for communicating with the prover.

The way in which this prover incorporates user interaction is the characteristic that makes it particularly well suited for proving verification conditions derived from real programs. The prover is based on natural deduction, as opposed to a "less natural" process such as resolution. For example, deductions are carried out directly in terms of the operations given in the verification condition rather than in terms of aquivalant clauses composed of ANDs, ORs, and NOTs. When the human user desires to interact with the prover, the dialogue is expressed in terms that are natural and convenient for the human instead of in those more convenient for a computer. In other words, the computer supports the human rather than vice versa.

The interactive policy of the prover is based on the premise that if the prover can construct a proof automatically, it will do it fairly quickly. For each theorem or subgoal, a time limit is set; if a proof has not been constructed in that time, the prover stops and waits for interactive direction. The user then has available a number of commands for displaying the theorem and the details of what the prover has done so far. Using these commands, the user isolates the difficulty and then can allocate more time, direct the prover into a new line of re[illegible]ing, supply additional information, or simply assume that the current subgoal is true and go on to another part of the proof. Typically, proofs of verification conditions will fail initially because they do not contain enough information for a rigorous proof. A very useful feature of the prover is that this additional information need not be stated initially, but rather can be supplied at the point in the proof when it is realized that this is necessary. This prevents the curious spectacle of the user having to prove the theorem himself before he asks the prover to do so, in order to determine what additional theorems and definitions will be needed.

To make the prover more useful in proving verification conditions, additional facilities have been added for handling relational expressions (involving <, LE, etc.) and proofs by cases. For variables that appear in relational expressions, both upper and lower bounds are computed. When a relational expression is discovered, as a hypothesis, the bounds on these variables are updated accordingly. This interval information represents the "state of the world" for these variables at that time and serves as additional hypotheses to the theorem or subgoal being considered. For example, if a contradiction such as J IN [K, K-1] occurs, this represents a false hypothesis and successfully terminates the proof. Also, if the bounds I IN [N+1, INFINITY] are already established and the hypothesis I LE N+1 is discovered, the updated bound is I IN [N+1, N+1], and this is treated as the equality I = N+1. These bounds are used not only to prove conclusions that are relational expressions, but also, more importantly, to give *partial results* that initiate proofs by cases. For example, suppose we have a theorem of the form

```
          H
      AND (I IN [K+1, N] IMP C)
    IMP C
```

and we know that I IN [K, N]. The prover will recognize that it can establish a partial result for the theorem in the case for I IN [K+1, N]. Then, by comparing the interval [K+1, N] with the bound [K, N] for I, it will conclude that the remaining case is I = K and attempt to reprove the theorem for that case. This type of situation frequently arises in proving assertions that are loop invariants.

The following summary of the proof of the three simplified verification conditions in Figure 3 indicates the general character of the prover and its policies toward interaction. A detailed view of the prover (being applied to topology theorems rather than verification conditions) is given by Bledsoe and Bruell [1973], and a complete description of variable bounding methods and proofs by cases, which have been added for program verification, is given by Bledsoe and Tyson [1974a,b].

SVC1, SVC2, and SVC3 are passed to the prover one at a time at the discretion of the user. For SVC1 there is nothing the prover (or a human prover, for that matter) can do without further information about APERM. What APERM(A,A,1,N) says is that A[1 . . . N] is a permutation of A[1 . . . N], and the user must recognize that this is a universal fact about APERM. This problem domain fact is given to the prover in the form of a reduction rule similar to those used by the symbolic evaluator for built-in operations:

(R1)* APERM(A,A,1,N) reduces to TRUE.

The actual sequence of events is (a) SVC1 is passed to the prover, (b) it tries to prove it and fails, (c) the user supplies the new reduction rule, and (d) the prover tries again and succeeds.

SVC2 comes from the loop of the SORT function, and its proof is the most involved. The prover first automatically breaks the proof into two subgoals, one for conclusion C1 and the other for C2. The prover fails on C1 until the user supplies an additional hypothesis that gives conditions under which swapping array elements preserves permutation:

```
              APERM(B,A,R,S)
          AND R LE X AND X LE S
          AND R LE Y AND Y LE S
(AH1)     IMP APERM(ASWAP(B,X,Y),A,R,S).
```

------------

*R1 is stated directly in terms of A to improve readability. We could just as well have used APERM(X,X,Y,Z). This same approach is used in this paper for subsequent inputs to the prover.

Once this is given, the prover realizes that it could use the conclusion of AH1 to prove the main conclusion C1 if it can prove the hypothesis of AH1, which amounts to

```
                APERM(B,A,1,N)
         AND 1 LE LOCMAX(B,1,K) AND LOCMAX(B,1,K) LE N
         AND 1 LE K AND K LE N.
```

The prover automatically proves these new subgoals from H5 and from H1-H4 which have been co verted to the form

```
         LOCMAX(B,1,K) IN [1,K]
         N IN [K, INFINITY]
         K IN [MAX(2, LOCMAX(B,1,K)), N].
```

Thus, the proof of C1 is done automatically with the addition of one user-supplied hypothesis.

The proof of conclusion C2 requires two different, user-supplied hypotheses. With the addition of these two, AH2 and AH3, the theorem to be proved becomes

```
(AH2)    ALL C,U,X,Y (U NE X AND U NE Y IMP ASWAP(C,X,Y)[U] = C[U])
(AH3)    AND ALL C,U,V,X,Y (U LE X AND X LE V AND U LE Y AND Y LE V
                 IMP AMAX(ASWAP(C,X,Y),U,V) = AMAX(C,U,V))
(H1)      . ND 1 LE LOCMAX(B,1,K)
(H2)     AND LOCMAX(B,1,K) LE K
(H3)     AND K LE N
(H4)     AND 2 LE K
(H5)     AND APERM(B,A,1,N)
(H6)     AND ALL I (K+1 LE I AND I LE N  IMP B[I] = AMAX(B,1,I))
(H7)     AND B[LOCMAX(B,1,K)] = AMAX(B,1,K)
```

```
(C2)    IMP ALL I (K LE I AND I LE N)
                IMP ASWAP(B,LOCMAX(B,1,K),K)[I]
                = AMAX(ASWAP(B,LOCMAX(B,1,K),K),1,I).
```

(Actually, the quantified variables are replaced by Skolem variables, but that need not concern us here.) AH2 gives conditions under which array subscripting is insensitive to swapping elements, and AH3 gives similar conditions for AMAX.

Once AH2 and AH3 are supplied, the prover eventually adopts a strategy of proof by cases, one case for I IN [K+1, N] and the other for I = K. The proof for I IN [K+1, N] requires establishing a chain of equalities involving the conclusions of AH2, AH3, H6, and C2. Although the prover has the machinery for building this chain, it does not invoke it automatically because the chain-building process is combinatorially explosive. The user, however, can explicitly direct it to try to build an equality chain. Once the user so directs, it builds the chain noting the hypotheses of AH2 and AH3 that must be proved in order that the chain be valid. These hypotheses are

```
        I NE LOCMAX(B,1,K)
    AND I NE K
```

from AH2,

```
        1 LE LOCMAX(B,1,K) AND LOCMAX(B,1,K) LE I
    AND 1 LE K AND K LE I
```

from AH3, and

```
        K+1 LE I AND I LE N
```

from H6. These conditions are required to hold only for the case I IN [K+1, N] and are easily proved automatically.

The proof of C2 for the case I = K begins with the automatic substitution of K for I. C2 then becomes

ASWAP(B, LOCMAX(B,1,K),K)[K]

= AMAX(ASWAP(B,LOCMAX(B,1,K),K),1,K)

which automatically is reduced to

(C2') B[LOCMAX(B,1,K)]

= AMAX(ASWAP(B,LOCMAX(B,1,K),K),1,K)

by a call to the symbolic evaluator. Again the prover is interactively directed to try an equality chain, which it succeeds in building using C2', AH3, and H7. Once again the chain is conditional, requiring from the hypothesis AH3 the new subgoals

1 LE LOCMAX(B,1,K) AND LOCMAX(B,1,K) LE K

AND 1 LE K AND K LE K.

These are proved automatically and the proof of SVC2 is complete.

The proof of SVC3 begins with the automatic substitution of 1 for K, giving

```
        1 LE N
    AND APERM(B,A,1,N)
    AND ALL I (2 LE I AND I LE N IMP B[I] = AMAX(B,1,I))
  IMP ALL I (1 LE I AND I LE N IMP B[I] = AMAX(B,1,I)).
```

The prover then automatically considers two cases, I IN [2,N] and I = 1. The I IN [2,N] case follows immediately, and the I = 1 case requires only the user-supplied reduction rule

(R2) AMAX(B,X,X) reduces to B[X],

and the proof of SVC3 is complete.

It should be noted that although some interaction has been necessary to direct the prover in its selection of strategy, most of the interaction was used to supply three additional hypotheses and two new reduction rules about APERM and AMAX. This amount of interaction seems quite tolerable considering that no information whatsoever about either APERM or AMAX was available to the prover, or any other part of the system, prior to the interaction. We anticipate that this example is typical of real programs in this regard, and that the main use of interaction will be to supply the prover with these additional facts about the problem domain.

## CONCLUSION

This has been an initial progress report on the development of an interactive program verification system whose ultimate goal is to be an effective tool in proving programs that solve significant, real problems. Certainly, the sorting program proved here is not a large, complex program. Nevertheless, we believe that its proof is, in two significant ways, typical of the proofs that will be necessary for much larger programs. First, we believe that, if we are to be able to prove large programs, such programs *and their specifications* must be expressed in sufficiently abstract terms so that their proofs can be carried out in terms of intellectually manageable segments. Although the principles of abstraction, such as those discussed by Dahl, Dijkstra, and Hoare [1972], seem now to be fairly well accepted for programs, little has been said of the need for abstraction in specifications. To keep the proof at an abstract level, though, clearly both are necessary. The APERM, AMAX, and ASWAP functions of the sort program are small, but representative, initial steps in this direction. Even these minor abstractions keep the proof at a higher, more abstract level, thus making it more succinct though no less rigorous. Second, we believe that proofs of large programs will require information about the domain of the problem being solved, which information will not be stated explicitly either in the program or its specifications. Currently, it seems highly desirable to allow this information to reside in the data base of the user's mind and to let him do the retrieval of the relevant facts as the need arises, as was done in supplying the additional hypotheses and reduction rules for the sort proof.

The current system clearly needs considerably more work to make it an "effective tool" for proving large programs. Significant areas in which additional efforts are now being made include incorporation of more features of Pascal, inclusion of other languages, extension of the assertion language, additional ways of exploiting abstraction, improved syntax analysis facilities, reconsideration of the division of labor and degree of integration among the major system components, improved facilities for display of the proof, improvement of the interactive user-system interface, and additional facilities for managing complex proofs composed of proofs of a large number of small program segments. Although we now see many ways in which the system can be improved and are very enthusiastic about its eventual success, these insights and this optimism are due ultimately to our having been able to bring together several complex and sophisticated, but diverse, software components, and to use them in a highly flexible and dynamic experimental environment. It seems appropriate to close by recalling a portion of R. W. Hamming's 1968 ACM Turing Lecture:

> Indeed, one of my major complaints about the computer field is that whereas Newton could say, "If I have seen a little farther than others it is because I have stood on the shoulders of giants," I am forced to say, "Today we stand on each other's feet." Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way. Science is supposed to be cumulative, not almost endless duplication of the same kind of things. [Hamming 1969, p. 10]

## *REFERENCES*

W. W. Bledsoe and P. Bruell 1973. A man-machine theorem-proving system, ***Advance Papers of Third International Joint Conference on Artificial Intelligence***, 1973, 56-65. Also ***Artificial Intelligence***, *5*, 1, Spring 1974, 51-72.

W. W. Bledsoe and M. Tyson 1974a. Typing and proofs by cases in program verification (working titl ), University of Texas at Austin Mathematics Department Memo ATP 15 (forthcoming).

W. W. Bledsoe and M. Tyson 1974b. The sup-inf method in Presburger arithmetic, University of Texas at Austin Mathematics Department Memo ATP 18 (forthcoming).

O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare 1972. ***Structured Programming***, Academic Press, 1972.

D. I. Good 1974. Provable programming (forthcoming).

R. W. Hamming 1969. One man's view of computer science, ***J. ACM***, *16*, 1, Jan. 1969, 3-12.

A. C. Hearn 1971. Reduce 2: A system and language for algebraic manipulation, ***Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation***, ACM, 1971, 128-133. Also Reduce 2 user's manual, University of Utah UCP-19, second edition, 1974.

C. A. R. Hoare 1971. Procedures and parameters: An axiomatic approach, in ***Symposium on Semantics of Algorithmic Languages***, E. Engeler, ed., Springer-Verlag, 1971, 102-116.

C. A. R. Hoare and N. Wirth 1973. An axiomatic definition of the programming language Pascal, ***Acta Informatica***, *2*, 335-355.

S. Igarashi, R. L. London, and D. C. Luckham 1973. Automatic program verification I: A logical basis and its implementation, USC Information Sciences Institute Report ISI/RR-73-11, May 1973. Also ***Acta Informatica***, 1974 (forthcoming).

R. L. London and D. R. Musser 1974. The application of a symbolic mathematical system to program verification, ***Proceedings of ACM Annual Conference***, 1974 (forthcoming).

L. C. Ragland 1973. A verified program verifier, Ph.D. thesis, University of Texas at Austin, 1973.

R. J. Waldinger and K. N. Levitt 1973. Reasoning about programs, ***Conference Record of ACM Symposium on Principles of Programming Languages***, 1973, 169-182.

Y.-Y. L. Wang 1973. A Nucleus verification condition compiler, M.S. thesis, University of Texas at Austin, 1973.